A Concise Guide to Clean 3.x StdEnv

Peter Achten
Radboud University, Netherlands
version november 2018
P.Achten@cs.ru.nl

November 19, 2018

Abstract

These notes serve as a brief guide to the Clean 3.x StdEnv, the set of library modules that come with the standard distribution. It is assumed that the reader has basic knowledge of programming in Clean.

1 Roadmap

Here is a brief overview of all Clean StdEnv modules in alphabetic order and in which section you can find its explanation.

module	section	page	module	section	page
`StdArray`	6	19	`StdBool`	4.1	6
`StdChar`	4.4	6	`StdCharList`	7	21
`StdClass`	2.2	3	`StdDebug`	12	24
`StdEnum`	5.1	12	`StdFile`	4.5	7
`StdFunctions`	9	22	`StdInt`	4.2	6
`StdList`	5	11	`StdMaybe`	10	22
`StdMisc`	11	24	`StdOrdList`	5.5	16
`StdOverloaded`	2.1	1	`StdOverloadedList`	5.6	17
`StdReal`	4.3	6	`StdString`	7	21
`StdTuple`	8	21			

2 An overloaded API

In order to appreciate the Clean StdEnv, we first need to have a look at the role of overloading. Two modules play a key role: `StdOverloaded` and `StdClass`.

2.1 StdOverloaded

This module defines all standard overloaded functions and operators. For historic reasons, the keyword used for overloading is *class*, which must not be confused with object oriented

classes. The name is really an abbreviation for *type class*, or, even longer, *type constructor class*. In `StdOverloaded` you find the usual suspects, such as arithmetic (`+`, `-`, `*`, /, `^`, `abs`, `sign`, `~`) and comparison operations (`<`, `<=`, `==`, `>=`, `>`).

Clean does no automatic *coercion* of values: if an `Int` value is expected, then an `Int` value should be provided, and not a `Real` or a `Bool` (even if you can think of sensible coercions). As an example, `1 + 2.5` does not type check in Clean. These kinds of coercions need to be done explicitly by you, and for this purpose `StdOverloaded` provides two families of coercion functions, viz.:

```
class toInt    a :: !a      ->  Int       // Convert into Int
class toChar   a :: !a      ->  Char      // Convert into Char
class toBool   a :: !a      ->  Bool      // Convert into Bool
class toReal   a :: !a      ->  Real      // Convert into Real
class toString a :: !a      ->  {#Char}   // Convert into String
```

for coercing a value of some type to the given basic type, and

```
class fromInt    a :: !Int     -> a       // Convert from Int
class fromChar   a :: !Char    -> a       // Convert from Char
class fromBool   a :: !Bool    -> a       // Convert from Bool
class fromReal   a :: !Real    -> a       // Convert from Real
class fromString a :: !{#Char} -> a       // Convert from String
```

for coercing a basic type to a desired type. The earlier example can be transformed to the following variants:

- `toReal 1 + 2.5` (which yields `3.5` of type `Real`)
- `fromInt 1 + 2.5` (which yields `3.5` of type `Real`)
- `1 + toInt 2.5` (which yields `3` of type `Int`)
- `1 + fromReal 2.5` (which yields `3` of type `Int`)

The overloaded constants `zero` and `one` are usually used in combination with the arithmetic operators (`+`, `-`, `*`, and /). The `zero` and `one` instances should adhere to the usual algebraic laws that you know from high school:

$$\begin{aligned} \texttt{zero} + x &= x = x + \texttt{zero} \\ &\quad\; x = x - \texttt{zero} \\ \texttt{one} * x &= x = x * \texttt{one} \\ &\quad\; x = x/\texttt{one} \end{aligned}$$

Clean cannot enforce this, so you should make certain that these basic properties hold for your custom defined instances.

Finally, one overloaded function is defined differently from all others:

```
class length m  :: !(m a) -> Int
```

This is an example of a *type constructor class*. It says that `length` is overloaded for a *type constructor* `m` that can hold values of type `a`. StdEnv contains only *one* instance of this overloaded function: in `StdList` the type constructor is the list type (`[]`), and the instance computes the length of a list (see also section 5.4).

2.2 StdClass

When you are developing overloading functions (i.e. functions that use overloaded operators and functions, and hence become overloaded themselves), you quickly notice that these functions use similar groups of overloaded functions. A typical example is `+`, `-`, `zero`. A function that uses these operators would normally have the following signature:

```
my_overloaded_function :: ... a ... | +, -, zero a
```

Instead of enumerating every single overloaded function or operator, you can use:

```
my_overloaded_function :: ... a ... | PlusMin a
```

Here, `PlusMin` is not an overloaded function, but a *collection* of overloaded functions or operators, and is defined in `StdClass`:

```
class PlusMin a | + , - , zero a
```

Such a group of overloaded functions can be used in the overloaded context restriction of a function. It automatically expands to the single member functions. For this reason, the same keyword *class* is used.

Another frequently occurring group of overloaded functions is `*`, `/`, `one`. For this, another group is defined in `StdClass`:

```
class MultDiv a | * , / , one a
```

(Type constructor) classes can be combined to form larger classes. An example is the following, that you can also find in `StdClass`:

```
class Arith a | PlusMin , MultDiv , abs , sign , ~ a
```

An overloaded function that uses (a subset of) these overloaded functions can have signature:

```
my_overloaded_function :: ... a ... | Arith a
```

which is much shorter than:

```
my_overloaded_function :: ... a ... | +, -, zero, *, /, one, abs, sign, ~ a
```

The classes `IncDec` and `Enum` are used to create lists with the *dot-dot* notation (see also section 5.1).

The class `Eq` contains the overloaded equality test operator `==`, and uses it to add the inequality test `<>`, which is defined in terms of `==` in the obvious way:

```
class Eq a | == a
where
  (<>) infix 4 :: !a !a -> Bool | Eq a
  (<>) x y :== not (x == y)
```

Here, `<>` is a derived member of the `Eq` class.

Finally, the `Ord` class derives a number of useful operators once `<` is given, viz. `>`, `<=`, `>=`, `min` and `max`. This makes sense only if `<` is transitive: if a `<` b *and* b `<` c, *then* a `<` c.

3 Do I have to worry about !, *, ., and u:?

Clean types are *annotated*, i.e. there is additional information attached to the type of functions, operators, and data types. There are two sorts of annotations: *strictness* (`!`) and *uniqueness* (`*`, `.`, `u:`). To understand the *bare* type of functions and operators, you usually do not need to know the meaning of these annotations, and can therefor safely ignore them. There are three notable exceptions: when working with files (section 4.5), using list overloading (section 5.6), and when working with arrays (section 6). The annotations convey information of the behavior of a function that cannot be derived from its bare type only.

3.1 Strictness annotations

The strictness annotation is the `!` symbol, prefixed immediately before the annotated type. Clean is, by default, a *lazy* language: it evaluates a computation only if it is really needed to obtain the result of the program. Usually, a computation *becomes* needed when it is passed as an argument to a function that in one way or another needs to know (part of) the *value* of that computation. For instance, to compute the sum of two computations, surely `+` requires the values of both its arguments. Another example: to take the head element of a list, surely `hd` must inspect the beginning of the list data structure that it is applied to. This is a property of the function, not of its type. Because types are used to communicate properties of functions in *definition modules*, they are suited candidates to *piggy back* this information, and therefor we annotate the types of functions:

```
class (+) infixl 6 a :: !a !a -> a      // in StdOverloaded
instance + Int                          // in StdInt
hd :: ![a] -> a                         // in StdList
```

By declaring an instance of `+` for type `Int`, you really make the following function available:

```
(+) infixl 6 :: !Int !Int -> Int
```

Both arguments are annotated with `!`, and hence you know that `+` will have to *evaluate* both arguments in order to compute a sum. Similar, `hd` is defined as:

```
hd [a:x]    = a
hd []       = abort "hd␣of␣[]"
```

The *pattern match* on the list argument forces the evaluation of any argument passed to `hd` to the structure of the first list constructor, *but no further*. Therefor, it is safe to compute:

```
Start = hd [42 : abort "That␣hurt!"]
```

3.2 Uniqueness annotations

Clean is a *pure* and *lazy* functional language. To be more precise, it is a pure, lazy *graph rewriting* language, i.e.: it allows *sharing* of arbitrary (sub)computations. This is great from a language engineers point of view because it allows easier reasoning as well as a host of optimizing language transformations. However, can a language that is pure (no assignments)

and lazy (no control flow) and that shares computations (long living computations) do interesting things such as file I/O? graphical I/O? use memory efficiently? There is a long answer and a short answer, and I'll stick to the short answer here: yes, they can.

A slightly longer answer is: a function can update an argument *in place* (which is an assignment) *if only* it knows for sure that it is the only piece of code in the program *at the point of evaluation* that has access to that argument. Put in other words: if a function has *unique* access to a data structure, then it can reuse the memory of that data structure without harming any of the highly desirable properties of being pure, lazy and shared.

A function can handle its argument(s) uniquely, i.e.: it does not introduce sharing on that argument. This is a property of the function, and is always inspected by the *uniqueness analysis* of the Clean compiler. However, it is not enough that the function treats this argument uniquely to allow updates on that argument, it must also be certain that that argument *is* unique at every time that the function is called. This can of course not be guaranteed by the function, but only by the calling party. To let them know that this function can update its argument, it annotates the type of that argument with the uniqueness attribute, `*`. This states that the function *will always* update its argument. Examples are opening and closing files, writing data to file, opening windows and menus in a GUI program. In case of *polymorphic* arguments, a function can tell its environment that it does *not change* the uniqueness of that argument by putting the `.` annotation in all occurrences of type argument in its type. This means that you can call the function with either a unique argument or a shared argument. An example is the `hd` function from `StdList`:

```
hd :: ![.a] -> .a
```

You can give it a list that contains elements with shared computations (`[a]`) but also a list in which none of the elements have shared computations (`[*a]`). In either case, `hd` returns the first element of that list, without changing its uniqueness property.

In some cases, these uniqueness dependencies require that you can *name* them so that you can refer to them in other uniqueness constraints. In that case, you can use a name, say `u`, and attach it to a type with `u:`. There are many examples in `StdList`. One such function is `tl`:

```
tl :: !u:[.a] -> u:[.a]
```

This type says that you can apply `tl` to both a list with shared computations as well as a list without shared computations, but the uniqueness of the result list is exactly the same (namely `u`) as that of the argument list.

4 Modules for basic types

Now that we have dealt with overloading and (type constructor) classes as well as annotations (`!`, `*`, `.`, `u:`), we can inspect the modules that define operations on the *basic types* of Clean. Clean offers six basic types: `Bool`, `Int`, `Real`, `Char`, `File`, and `String`. The `String` type is a bit special because it is really a composite type (array of unboxed `Char`s, to be precise) and is used together with the `StdArray` module, as well as lists, so I treat `String`s separately in Section 7. For each basic type *Type*, a separate module named `Std`*Type* exists. They consist mostly of *instance declarations* of the overloaded functions that are defined in the `StdOverloaded` module that was discussed in Section 2.

4.1 StdBool

StdBool is the smallest of the basic type modules. Besides a few overloaded operations (== and coercion), it provides the usual boolean operators && (conditional and), || (conditional or), and not (negation). Note that the types of && and || suggest that these operations are *conditional*:

```
(||) infixr 2 :: !Bool Bool -> Bool
(&&) infixr 3 :: !Bool Bool -> Bool
```

because the second argument has no strictness annotation ! (see Section 3.1). If these operations weren't conditional, then they would have to evaluate both arguments in all cases, and therefor would be strict in both arguments.

4.2 StdInt

StdInt implements many of the arithmetic overloaded operations that you encounter in StdOverloaded. Besides these, it also allows you to do bitwise manipulation of integer numbers. These are:

```
(bitor)  infixl 6 :: !Int !Int -> Int
(bitand) infixl 6 :: !Int !Int -> Int
(bitxor) infixl 6 :: !Int !Int -> Int
(<<)     infix  7 :: !Int !Int -> Int
(>>)     infix  7 :: !Int !Int -> Int
bitnot             :: !Int     -> Int
```

4.3 StdReal

StdReal is very similar to StdInt except that it does not offer bit manipulation operations, but instead gives you access to the usual trigonometry operations, as well as raising powers, taking logarithms, and computing the square root.

4.4 StdChar

StdChar defines instances for basic computations and coercion on ASCII characters. Clean does not support Unicode. A few additional coercion functions are defined on the *value* of their Char argument:

```
digitToInt :: !Char -> Int   // Convert Digit into Int
toUpper    :: !Char -> Char  // Convert Char into an uppercase Char
toLower    :: !Char -> Char  // Convert Char into a lowercase Char
```

(digitToInt c) yields the integer value of the digit representation of that value if c ∈ {'0' ... '9'}. (toUpper c) yields the upper case character if c ∈ {'a' ... 'z'}, and c otherwise. (toLower c) yields the lower case character if c ∈ {'A' ... 'Z'}, and c otherwise.

StdChar also defines a number of predicates on Char values, that are useful when working with texts.

```
isUpper    :: !Char -> Bool    // True if arg1 is an uppercase character
isLower    :: !Char -> Bool    // True if arg1 is a lowercase character
isAlpha    :: !Char -> Bool    // True if arg1 is a letter
isAlphanum :: !Char -> Bool    // True if arg1 is an alphanumerical character
isDigit    :: !Char -> Bool    // True if arg1 is a digit
isOctDigit :: !Char -> Bool    // True if arg1 is a digit
isHexDigit :: !Char -> Bool    // True if arg1 is a digit
isSpace    :: !Char -> Bool    // True if arg1 is a space, tab etc
isControl  :: !Char -> Bool    // True if arg1 is a control character
isPrint    :: !Char -> Bool    // True if arg1 is a printable character
isAscii    :: !Char -> Bool    // True if arg1 is a 7 bit ASCII character
```

4.5 StdFile

StdFile is the only module in StdEnv that deals with the 'impure' external world. Clean does offer a lot of other libraries for these kinds of operations, but they are not part of StdEnv. The external world is known by the type World. There is only one world, so most functions expect that single world and update it. Hence, their type signature is something like ... *World ... -> ... *World .... The functions in StdFile are grouped into the following categories: managing files, working with read-only files, and working with read-write files. These are explained below.

4.5.1 Managing files

In order to work with a file, you need to *open* it, and *close* it when you're done. These file management functions are collected within one (type constructor) class, called FileSystem:

```
class FileSystem env where
    fopen  :: !{#Char} !Int !*env -> (!Bool,!*File,!*env)
    fclose :: !*File        !*env -> (!Bool,       !*env)
    stdio  ::               !*env -> (      !*File,!*env)
    sfopen :: !{#Char} !Int !*env -> (!Bool,! File,!*env)
```

In StdFile, two instances of this class are provided. The only relevant instance is of type World, the other (Files) is only present for historic reasons (I can tell you about it if you really want to know). So we have:

```
instance FileSystem World
```

which gives us the following functions to play with:

```
fopen  :: !{#Char} !Int !*World -> (!Bool,!*File,!*World)
sfopen :: !{#Char} !Int !*World -> (!Bool,! File,!*World)
stdio  ::               !*World -> (      !*File,!*World)
fclose :: !*File        !*World -> (!Bool,       !*World)
```

There are three functions to *open* a file: fopen, sfopen, and stdio. The difference between fopen and sfopen is that fopen opens a file that can be written to and read from (it is updated, hence it must be unique), and that sfopen opens a read-only file (it can not be updated, hence

it need not be unique). The function `stdio` opens a special file, *stdio*, that connects with the console for simple line-based I/O programs (it is updated, hence it must be unique). Only the unique files need to be closed when you're done with them. This is done with `fclose`. Note that if you accidently forget to close a unique file, you can not be certain that all data is actually written to that file due to internal buffering. It is also impossible to reopen that file within the same program for further processing.

The first argument of `(s)fopen` is the *file name*. If you simply pass the name of a file, without any directory path before it, then this works on the current directory of the application. If the file name also consists of a directory path, then that location is used. The second argument of both functions controls the *file mode*. This is an integer value, and should be one of:

```
FReadText   :== 0   // Read from a text file
FWriteText  :== 1   // Write to a text file
FAppendText :== 2   // Append to an existing text file
FReadData   :== 3   // Read from a data file
FWriteData  :== 4   // Write to a data file
FAppendData :== 5   // Append to an existing data file
```

Use on of the first three modes if you wish to work with ASCII files, and use one of the last three modes if you wish to work with *binary* files. If you want to *read* either file, use `FRead`*Mode*; if you want to clear the file content and *write* new data, use `FWrite`*Mode*; and if you want to *extend* the current content, use `FAppend`*Mode*. The third parameter must be the current world, of which there is only one anyway.

A few examples: to open a text file, called `"test.txt"`, and read its content:

```
# (ok,file,world) = fopen FReadText "test.txt" world
```

To add data to a log file in some standard directory:

```
# (ok,file,world) = fopen FAppendText
                          "C:\\Program␣Files\\MyProgram\\settings.log"
                          world
```

`(s)fopen` returns a boolean to report success (`True` only if the file could be opened), the actual file, and the updated world in which the file has been opened. The file result can only be used if the file could be created. Any attempt to use it results in a run-time error, so check your boolean! Hence, typical idiom is:

```
# (ok,file,world) = fopen ...
| not ok          = abort "The␣file␣could␣not␣opened.\n"
... // now you know it is safe to use file
```

or, if you dislike `#`:

```
case fopen ... of
    (False,_,  world) = ... // opening failed, continue with world
    (True,file,world) = ... // opening succeeded, safe to use file
```

The typical structure of a basic file manipulating program is:

```
Start :: !*World -> *World
Start world
# (ok,file,world) = fopen mode name world
| not ok          = abort ("Could␣not␣open␣file␣'" +++ name +++ "'.")
# file            = do_something_interesting_with file
# (ok,world)      = fclose file world
| not ok          = abort ("Could␣not␣close␣file␣'" +++ name +++ "'.")
| otherwise       = world
where
    mode          = // file mode of your choice
    name          = // file name of your choice
```

Finally, there is one function that allows you to change the *file mode* of a file that has already been opened with `fopen`:

```
freopen :: !*File !Int -> (!Bool,!*File)
```

The integer argument is the new file mode that the file should have. Again, the boolean returns `True` only if this operation was successful, and the new file value can only be used safely if that was the case.

4.5.2 Read-only files

Read-only files are created with `sfopen`, as explained above. Because they are not updated, they can be shared safely, and hence do not require to be unique. Their type is therefor just `File`.

The following read operations are available on read-only files:

```
sfreadc     :: !File          -> (!Bool,!Char,    !File)
sfreadi     :: !File          -> (!Bool,!Int,     !File)
sfreadr     :: !File          -> (!Bool,!Real,    !File)
sfreads     :: !File !Int     -> (      !*{#Char},!File)
sfreadline  :: !File          -> (      !*{#Char},!File)
sfend       :: !File          -> Bool
```

To read the next, single, `Char` (`Int`, `Real`) value from the file, use `sfreadc` (`sfreadi`, `sfreadr`). Because reading advances the internal *read pointer* of the file, these functions need to return a new file value. As usual, the boolean result indicates successful reading.

To read a maximum of `n Char` values from `file`, use `sfreads file n`. To read the next line from `file`, use `sfreadline`. A line is a sequence of `Char`s, returned as a `String`, terminated by a newline character. If the end of the file was not terminated with a newline character, then the `String` simply contains the remainder of the file content. You can test whether you've reached the end of `file` with `sfend file`.

The above functions read a file from start to end. In some cases you want to *seek* a file, i.e. start at specific positions within a file. It is sometimes useful to determine the current value of the internal *read pointer* of a file. This is done with `sfposition file` which returns that value.

```
sfposition :: !File -> Int
```

The internal *read pointer* can be set with `sfseek` which expects three arguments: the first is the file itself and the second and third argument determine the new value of the internal *read pointer*. This third argument determines from which relative position the second argument, an *offset*, should be interpreted:

FSeekSet: set read pointer to *offset*;

FSeekCur: advance read pointer *offset* characters;

FSeekEnd: advance read pointer *offset* characters from the end of the file.

```
sfseek   :: !File !Int !Int -> (!Bool,!File)

FSeekSet :== 0   // New position is the seek offset
FSeekCur :== 1   // New position is the current position plus the seek offset
FSeekEnd :== 2   // New position is the size of the file plus the seek offset
```

4.5.3 Read-write files

Read-write files support the same read operations as read-only files do. The names of these functions are the same as of the read-only operations with the prefix `s` skipped. The types of the file, which can be updated, is `*File`. Hence, the operations that have been discussed above for read-only files appear similarly for read-write files (the only difference is that `fend` and `fposition` have to return a new file value):

```
freadc    :: !*File          -> (!Bool,!Char,    !*File)
freadi    :: !*File          -> (!Bool,!Int,     !*File)
freadr    :: !*File          -> (!Bool,!Real,    !*File)
freads    :: !*File !Int     -> (      !*{#Char},!*File)
freadline :: !*File          -> (      !*{#Char},!*File)
fend      :: !*File          -> (!Bool,          !*File)

fposition :: !*File          -> (      !Int,     !*File)
fseek     :: !*File !Int !Int -> (!Bool,         !*File)
```

Besides these operations, which are basically the same as for read-only files, read-write files offer one additional reading function:

```
freadsubstring :: !Int !Int !*{#Char} !*File -> (!Int,!*{#Char},!*File)
```

The meaning of `freadsubstring start end string file` is that `(end-start)` characters are read from `file` (if not possible, then less characters are read). Assume `n` characters are read. These `n` characters are *updated* in `string` (which is why it must be unique), starting at `start`, and ending at `start+n-1`. The result is `n`, the updated string, and the read file.

To write data to read-write files, use one of the following functions:

```
fwritec        ::          !Char    !*File -> *File
fwritei        ::          !Int     !*File -> *File
fwriter        ::          !Real    !*File -> *File
```

```
fwrites        ::          !{#Char} !*File -> *File
fwritesubstring :: !Int !Int !{#Char} !*File -> *File
```

To write a `Char` (`Int`, `Real`, `String`) to `file`, use `fwritec` (`fwritei`, `fwriter`, `fwrites`). To write the *slice* `s%(start,end)` to `file`, use `fwritesubstring start end s file`.

An alternative, and more concise, notation for writing data is the following overloaded operator:

```
class (<<<) infixl a :: !*File !a -> *File

instance <<< Char
instance <<< Int
instance <<< Real
instance <<< {#Char}
```

Its instance implementations correspond with `fwrite(c,i,r,s)` respectively. Because `<<<` is left-associative, you can write multiple data in one go:

```
file <<< "This␣line␣has␣" <<< 4 <<< "␣words␣" <<< '.'
```

Robust programs should check whether file manipulations have succeeded. After you have written data to `file`, you can check whether this was successful with `ferror file`. Hence, typical idiom of its use is:

```
# file        = file <<< "This␣line␣has␣" <<< 4 <<< "␣words␣" <<< '.'
# (fail,file) = ferror file
| fail        = abort "Could␣not␣write␣data␣file."
... // now you know it is safe to use file
```

If you have created a read-write file, and are completely satisfied, then you can turn it into a read-only file with `fshare :: !*File -> File`. Note that you cannot open this file any longer within the program for further writing.

Finally, there is one special read-write file that is particularly useful for writing *trace statements*:

```
stderr :: *File
```

It is special because it creates a read-write file that can be opened arbitrarily many times in a program. This is not possible for any of the other read-write files, which need to be closed before they can be opened. Note that in the standard environment, a more convenient *tracing* module is provided, viz. `StdDebug` (section 12).

5 Working with lists

Lists are the workhorses of functional programming languages. The language Clean is no exception to this rule. There is a lot of support for list operations, both in the language as well as in the standard environment (in particular the modules `StdList.dcl`, `StdOrdList.dcl`, `StdOverloadedList`, `StdStrictLists`). This section presents this support.

5.1 List notation

In order to work with lists, you first need to *denote* them. The simplest way of denoting lists is to enumerate all elements. Here are a number of examples:

```
Start = ( []                                    1.
        , [1,2,3,4,5,6,7,8,9,10]                2.
        , [1..10]                               3.
        , ['a'..'z']                            4.
        , ['a', 'c' .. 'z']                     5.
        , [10.0, 9.0 .. 0.0]                    6.
        )                                       7.
```

The simplest list of all is the *empty* list (line 1) which contains no elements. The type of an empty lists cleanly expresses that we do not know its element type: `[] :: [a]`. In line 2 a list is created with 10 elements, viz. 1 upto 10 (inclusive). If there is some sort of regularity, then it is much more concise to use the `..` notation (*dot-dot* notation). Examples are in lines 3 – 6. The list on line 3 has the same value as the one on line 2, and both are of type `[Int]`. The list on line 4 has value

```
['a','b','c','d','e','f','g','h','i','j','k','l','m'
,'n','o','p','q','r','s','t','u','v','w','x','y','z']
```

and type `[Char]`. A less cumbersome notation for `Char`-lists is:

```
['abcdefghijklmnopqrstuvwxyz']
```

which has exactly the same value as above. The examples on lines 5 and 6 shows that lists can also be enumerated that have a regular 'gap' between successive elements. The size of the gap is determined by computing the difference between the first two elements. Next elements are computed by keeping adding the gap until the border value has been reached. The list on line 5 has value

```
['acegikmoqsuwy']
```

and type `[Char]` and the list on line 6 has value

```
[10.0, 9.0, 8.0, 7.0, 6.0, 5.0, 4.0, 3.0, 2.0, 1.0, 0.0]
```

and type `[Real]`.

If you use *dot-dot* notation, you are required to import the module `StdEnum`. This module is automatically imported if you import `StdEnv`. `StdEnum` contains the functions that create the list when *dot-dot* notation is used. The Clean compiler transforms these expressions to these functions. This also explains why it is possible to use *dot-dot* notation for types defined by yourself, as soon as you have defined proper instances of the corresponding class `Enum`.

5.2 ZF-expressions

A powerful way to create and manipulate lists are *ZF-expressions*, or *list comprehensions*. The name *ZF-expressions* comes from the *Zermelo Frank* notation for sets in mathematics.

If you have a Clean predicate `isPrimeNumber :: Int -> Bool` which determines if a number is *prime*[1], then the list of prime numbers can be created by:

```
all_primes :: [Int]
all_primes = [x \\ x <- [1..] | isPrimeNumber x]
```

Here, the expression `x <- [1..]` is a *generator*. In this case it is the *infinite* list of integral numbers, starting with 1 and incremented by 1. The generator determines the subsequent value of `x`. Just as in function definitions, *conditions* are preceded by a `|` symbol. In this case the condition is `isPrimeNumber x`, for each `x` that is produced by the generator. Hence, the value of the function `all_primes` is an *infinite* list of integral numbers, in ascending order, starting from 1, that are also prime. The result of this function is therefor (at least, its beginning):

```
[1,2,3,5,7,11,13,17,19,23,
```

In a ZF-expression the result `x` can be manipulated as well. Suppose you need the set of squares of all prime numbers, then you can write this as follows:

```
all_prime_squares :: [Int]
all_prime_squares = [x*x \\ x <- [1..] | isPrimeNumber x]
```

With the other function definition, you could also have defined it as follows:

```
all_prime_squares :: [Int]
all_prime_squares = [x*x \\ x <- all_primes]
```

One remark on the use of infinite datastructures: in a functional language such as Clean it is no problem to define and manipulate them, as long as you don't try to compute them entirely. That will lead to a non-terminating program. The following function uses this property to compute the first hundred prime numbers:

```
first_100_primes :: [Int]
first_100_primes = take 100 all_primes
```

The standard function `take` n xs takes the first n elements of the list xs and has type `take :: Int [a] -> [a]` (see Section 5.4).

An (inefficient) way to create the function `isPrimeNumber` is the following ZF expression:

```
isPrimeNumber :: Int -> Bool
isPrimeNumber n = [1,n] == [divisor \\ divisor <- [1..n] | n rem divisor == 0]
```

This function examines too many cases to determine whether a number n is prime. However, it should be obvious that it correctly determines the desired property.

5.3 More ZF expressions

In the above section we have created lists with ZF expressions that are constructed with *generators* and *conditions*. It is also possible to combine generators in two ways:

1. If `as` and `bs` are lists, then the *Carthesian product* of both lists can be defined as:

[1] A number n is prime if its only integral divisors are the numbers 1 and n.

```
product as bs = [(a,b) \\ a <- as, b <- bs]
```

Assume that `as = [1,2,3]` and `bs = ['a','b']`, then:

```
product as bs = [(1,'a'),(1,'b'),(2,'a'),(2,'b'),(3,'a'),(3,'b')]
```

The length of the resulting list is equal to the product of the lengths of `as` and `bs`.

2. If `as` and `bs` are lists, then the elements of both lists can be combined element by element (*zipped*):

```
zipped as bs = [(a,b) \\ a <- as & b <- bs]
```

Assume that `as = [1,2,3]` and `bs = ['a','b']`, then:

```
zipped as bs = [(1,'a'),(2,'b')]
```

The length of the resulting list is the length of the shortest list.

All ways to create and manipulate ZF expressions that have been presented above can be combined in arbitrary ways. As a result, ZF expressions are flexible tools of expression to work with lists.

5.4 `StdList`: operations on lists

In this section the most frequently used functions of module `StdList` are discussed. These functions are grouped according to what you would like to do with lists.

5.4.1 I want to compare lists:

- as `==` bs: yields `True` only if as and bs have the same number of elements that also have the same value. Hence, `[`$a_1 \ldots a_n$`] == [`$b_1 \ldots b_m$`]` is `True` only if $n = m$ *and* $\forall 1 \leq i \leq n$: a_i `==` b_i.
- `isEmpty` as: yields `True` only if as = `[]`.

5.4.2 I want to know how many elements a list has:

- `length [`$a_1 \ldots a_n$`]` $= n$, and `length []` $= 0$.

5.4.3 I want to find an element of a list:

- `isMember` x $[a_0 \ldots a_n] =$ `True` only if $x = a_i$ for some $0 \leq i \leq n$.
- `isAnyMember` $[x_0 \ldots x_n]$ $as =$ `True` only if `isMember` x_i as for some $0 \leq i \leq n$.

5.4.4 I want to select an element of a list:

- `[`$a_0 \ldots a_n$`] !!` $i = a_i$.
- `hd [`$a_0 \ldots a_n$`]` $= a_0$.
- `last [`$a_0 \ldots a_n$`]` $= a_n$.

5.4.5 I want to build a shorter list:

- $[a_0 \ldots a_n]$ `%` $(i, j) = [a_i \ldots a_j]$ $(0 \leq i \leq j \leq n)$.
- `tl` $[a_0 \ldots a_n] = [a_1 \ldots a_n]$ $(n > 0)$.
- `init` $[a_0 \ldots a_n] = [a_0 \ldots a_{n-1}]$ $(n > 0)$.
- `take` k $[a_0 \ldots a_n] = [a_0 \ldots a_{k-1}]$ $(0 \leq k \leq n + 1)$.
- `takeWhile` p $[a_0 \ldots a_n] = [a_0 \ldots a_k]$ where $(p\ a_i) =$ `True` for all a_i in the resulting list, and $(p\ a_{k+1}) =$ `False`.
- `drop` k $[a_0 \ldots a_n] = [a_k \ldots a_n]$ $(0 \leq k)$.
- `dropWhile` p $[a_0 \ldots a_n] = [a_k \ldots a_n]$ where $(p\ a_i) =$ `True` for all $i < k$, and $(p\ a_k) =$ `False`.
- `filter` $p\ a =$ `[x \\ x <-` a `|` $p\ a$`]`.
- `removeAt` i $[a_0 \ldots a_n] = [a_0 \ldots a_{i-1}, a_{i+1} \ldots a_n]$.
- `removeMember` x as removes the first occurrence of x from as.
- `removeMembers` $[x_0 \ldots x_n]$ as removes the first occurrence of $x_0, x_1 \ldots x_n$ from as.
- `removeDup` as removes all duplicate elements from as (using `==`).
- `removeIndex` $e[x_0 \ldots x_{i-1}\ e\ x_{i+1} \ldots x_n] = (i, [x_0 \ldots x_{i-1}, x_{i+1} \ldots x_n])$, if $e \neq x_j$ voor $0 \leq j < i$.

5.4.6 I want to build a bigger list:

- $[a_0 \ldots a_m]$ `++` $[b_0 \ldots b_n] = [a_0 \ldots a_m, b_0 \ldots b_n]$.
- `flatten` $[l_0 \ldots l_n] = l_0$ `++` l_1 `++` $\ldots$ `++` l_n.
- `insertAt` $i\ x$ $[a_0 \ldots a_n] = [a_0 \ldots a_{i-1}, x, a_i \ldots a_n]$.
- `insert` $p\ x$ $[a_0 \ldots a_n] = [a_0 \ldots a_{i-1}, x, a_i \ldots a_n]$, such that $(p\ x\ a_i) =$ `True`, and for each $0 \leq j < i : \neg(p\ x\ a_j)$.

5.4.7 I want to modify a list, but retain the number of elements:

- `map` $f\ a =$ `[`f `x \\ x <-` a`]`.
- `reverse` $[a_0 \ldots a_n] = [a_n \ldots a_0]$.
- `updateAt` $i\ x$ $[a_0 \ldots a_n] = [a_0 \ldots a_{i-1}, x, a_{i+1} \ldots a_n]$.

5.4.8 I want to divide lists:

- `splitAt` i a = (`take` i a,`drop` i a).
- `span` p a = (`takeWhile` p a,`dropWhile` p a).
- `unzip` $[(a_0, b_0), (a_1, b_1) \ldots (a_n, b_n)] = ([a_0, a_1 \ldots a_n], [b_0, b_1 \ldots b_n])$.

5.4.9 I want to create a list:

- `iterate` f $x = [x, f\ x, f\ (f\ x), f\ (f\ (f\ x)) \ldots]$.
- `repeat` $x = [x, x, x \ldots]$.
- `repeatn` n x =`take` n (`repeat` x).
- `indexList` $[a_0 \ldots a_n] = [0 \ldots n]$.
- `zip` $([a_0, a_1 \ldots a_n], [b_0, b_1 \ldots b_n]) = [(a_0, b_0), (a_1, b_1) \ldots (a_n, b_n)]$.
- `zip2` $[a_0, a_1 \ldots a_n][b_0, b_1 \ldots b_n] = [(a_0, b_0), (a_1, b_1) \ldots (a_n, b_n)]$.

5.4.10 I have a list and want to reduce it to a single value:

- `foldl` f r $[a_0 \ldots a_n] = f\ (\ldots (f\ (f\ r\ a_0)\ a_1) \ldots)\ a_n$.
- `foldr` f r $[a_0 \ldots a_n] = f\ a_0\ (f\ a_1\ (\ldots (f\ a_n\ r) \ldots))$.

5.5 `StdOrdList`: operations on sorted lists

Being sorted is an important property for data structures. The module `StdOrdList` contains functions that manipulate sorted lists. The most frequently used functions are:

- `sort` *as* sorts the elements of *as* using the ordering `<` that is defined on values of the list element type.
- `merge` *as* *bs* = `sort` (*as* `++` *bs*) (but in a smarter way...). Important: `merge` assumes that *as* and *bs* are sorted.
- `maxList` *as* determines the maximum element of *as* using the ordering `<` that is defined on values of the element type of *as*. `maxList []` yields a run-time error.
- `minList` *as* determines the minimum element of *as* using the ordering `<` that is defined on values of the element type of *as*. `minList []` yields a run-time error.

Of each of the functions described above, `StdOrdList` also provides a ...`By` version, which is parameterized with a comparison function that is used instead of the `<` ordering on values of the list element type.

5.6 List overloading

In section 3.1 the role of the strictness annotation, `!`, was explained. Without annotation, lists are *lazy*, i.e.: when constructing a list neither the head element nor the tail list gets evaluated at that time[2], and this is true for any data structure in a lazy language such as Clean. This allows you to build arbitrarily large and complicated lists, knowing that the relevant parts will be evaluated when needed only. However, often you know in advance that either the list structure (*spine*) or the list elements or both will get evaluated anyway in your program. In these cases, it is more efficient to evaluate these parts while constructing the list. You can choose to evaluate the spine, the element, or even unbox the element (similar to `String` being an array of unboxed characters):

Spine strict list: a list of type `[a!]` is strict in the spine structure, hence it evaluates the tail at construction;

Element strict list: a list of type `[!a]` is strict in its elements, hence it evaluates the element at construction;

Unboxed list: a list of type `[#a]` evaluates `a` to normal form and stores the result in the list. This can only be done if `a` is a *basic type* (section 4).

This results in *five* additional, different types of lists besides the default lazy one of type `[a]`: `[a!]`, `[!a]`, `[!a!]`, and `[#a]`, `[#a!]`. While developing and testing your code, it is useful not to commit early on either one of these types, but direct this via the type signature of your functions: in this way you do not have to change your function implementations, but only their types. To do this, you need to use the `[|]` notation which basically states that the list you are constructing is going to be one of the above types. So, an overloaded *nil* is written as `[|]`, and an overloaded *cons* as `[|`x`:`xs`]`. The operations in `StdList` operate on lazy lists, and hence cannot be used on these new list types. For this reason, overloaded versions of these operations are defined in module `StdOverloadedList`. Their definitions might seem intimidating, but remember that they are basically the same as the ones found in `StdList`, except that their name starts with a capital letter, or has an additional symbol, and that they operate on `[|a]`, overloaded lists, instead of the default lazy lists `[a]`.

Example

The program in figure 1 illustrates the effect of using different versions of lists. In this example, the first two elements of the overloaded list `[|1,2,3,4,5]` are computed using the *six* versions of lists. The concrete type is controlled via the explicit type signature of the `Start` function. The list that is created is defined by the function `list` (line 21) that uses two constructor functions, `c` (line 22-25) and `nil` (line 26), whose sole purpose is to show a trace message when they are evaluated. The trace messages that are generated for the six versions of lists reflect the reduction behavior. For clarity, I underline the output of the `Start` function that emits the resulting value (the list [1,2]).

lazy list: (line 12) the traces follow the reduction behavior of the `Take` function and process only the first two elements of the list (shown without newlines):

[2]This concept hearkens back to the seminal paper *"CONS should not evaluate its argument"* by Friedman and Wise, presented at the *Third Colloquium on Automata, Languages and Programming* in 1976.

```
module list_overloading_example                                                  1
import StdEnv, StdDebug, StdOverloadedList                                       2
                                                                                 3
Start :: ( String, [ Int ]                                                       4
         , String, [!Int ]                                                       5
         , String, [ Int!]                                                       6
         , String, [!Int!]                                                       7
         , String, [#Int ]                                                       8
         , String, [#Int!]                                                       9
         , String                                                                10
         )                                                                       11
Start = ("\n[␣␣]:␣", Take 2 list                                                 12
        ,"\n[!␣]:␣", Take 2 list                                                 13
        ,"\n[␣!]:␣", Take 2 list                                                 14
        ,"\n[!!]:␣", Take 2 list                                                 15
        ,"\n[#␣]:␣", Take 2 list                                                 16
        ,"\n[#!]:␣", Take 2 list                                                 17
        ,"\n"                                                                    18
        )                                                                        19
                                                                                 20
list   = c 1 (c 2 (c 3 (c 4 (c 5 nil))))                                         21
c x xs = trace_n "<eval␣cons>"                                                   22
                    [| trace_n ("<eval␣" +++ toString x +++ ">") x               23
                     : trace_n "<eval␣tl>" xs                                    24
                    ]                                                            25
nil    = trace_n "<eval␣[]>" [|]                                                 26
```

Figure 1: Tracing all forms of list overloading

```
<eval cons>[<eval 1>1<eval tl><eval cons>,<eval 2>2]
```

element strict list: (line 13) in contrast with the lazy list, the traces show that when the cons node is evaluated, *also* the head element is evaluated:

```
<eval cons><eval 1>[1<eval tl><eval cons><eval 2>,2]
```

spine strict list: (line 14) in contrast with the lazy list, the traces show that when the cons node is evaluted, *also* the tail of the list is evaluated. As a consequence, the entire spine gets evaluated before `Take` can proceed:

```
<eval cons><eval tl><eval cons><eval tl><eval cons><eval tl><eval cons><eval tl>
<eval cons><eval tl><eval nil>[<eval 1>1,<eval 2>2]
```

strict list: (line 15) evaluates both the tail of the list and the element before `Take` can proceed:

```
<eval cons><eval tl><eval cons><eval tl><eval cons><eval tl><eval cons><eval tl>
<eval cons><eval tl><eval nil><eval 5><eval 4><eval 3><eval 2><eval 1>[1,2]
```

unboxed element list: (line 16) unboxing a value of basic type has the same evaluation order as the element strict list:

```
<eval cons><eval 1>[1<eval tl><eval cons><eval 2>,2]
```

unboxed element spine strict list: (line 17) unboxing a value of basic type, and evaluating the tail of a cons node as well, results in the same evaluation order as the strict list:

```
<eval cons><eval tl><eval cons><eval tl><eval cons><eval tl><eval cons><eval tl>
<eval cons><eval tl><eval nil><eval 5><eval 4><eval 3><eval 2><eval 1>[1,2]
```

6 Working with arrays

Clean supports *arrays*. Arrays differ from lists in several aspects:

- Arrays always have a *finite* number of elements, whereas lists can be infinitely long.
- Element selection is in constant time ($\mathcal{O}(1)$), whereas list element selection is linear ($\mathcal{O}(n)$).
- Arrays consume *contiguous* blocks of memory, whereas a list is a linked data structure.

In Clean, the type `String` is implemented as an array of characters. Hence, everything that is said here about arrays also applies to `String`s. For further manipulations of `String`s, I refer to section 7.

6.1 Array notation

Arrays are created by enumeration of their elements, in a similar way as with lists, except that the *delimiters* are `{` and `}` instead of `[` and `]`. So, `{}` denotes an array with zero elements, and `{1,2,3,4,5}` and array with five elements, viz. 1 up to 5.

Unfortunately, there is no *dot-dot* notation for arrays, so you can't write down `{1..10}` as you can for lists. However, if you happen to have a list, say `list = [1..10]`, then you can create an array with the same elements as follows:

```
{e \\ e <- list}
```

You can also use another array as a generator. Say you have `array = {1,2,3,4,5}`, then you can create another one:

```
{f e \\ e <-: array}
```

Note the different symbol for extracting elements from an array generator (`<-:`). Because you can use both arrays and lists as generators, you can easily transform lists into arrays and vice versa:

```
toList array = [e \\ e <-: array]
toArray list = {e \\ e <-  list}
```

Array creation is *overloaded*: the same expression, say `myArray = {1,2,3,4,5}`, can be one of three concrete types (I refer to the language manual for more details):

- `myArray :: {Int}`, which is a *lazy* array. This is basically an array of unevaluated element expressions.
- `myArray :: {!Int}`, which is a *strict* array. In such an array, all expressions are evaluated strictly.
- `myArray :: {#Int}`, which is an *unboxed* array. Here, the element type must be a basic type. In such an array, the element values are evaluated strictly and stored subsequently within the array.

6.2 Array manipulation

Arrays are overloaded, and this is also the case for its manipulation functions. Clean provides syntax for these operations, but internally they are translated to the functions that are imported via `StdArray` (in a similar way as those for *dot-dot* expressions with lists). Because unboxed arrays are only defined for basic types, there must be relation between the sort of array (lazy, strict, unboxed) and its element type. For this reason, the class `Array` that defines these operations is a true type constructor class:

```
class Array .a e where
    select      :: !.(a .e) !Int    -> .e
    uselect     :: !u:(a e) !Int    -> *(e, !u:(a e))
    size        :: !.(a .e)         -> Int
    usize       :: !u:(a .e)        -> *(!Int, !u:(a .e))
    update      :: !*(a .e) !Int .e -> *(a .e)
    createArray :: !Int e           -> *(a e)
    replace     :: !*(a .e) !Int .e -> *(.e, !*(a .e))
```

The array element selection operations *array*`.[`*index*`]` and *array*`![`*index*`]` are translated to `select` *array index* and `uselect` *array index* respectively. The zero-based *index* must be less than the size of the array. If you need an array for subsequent destructive updates, then you should use `uselect`, and otherwise you can use `select`.

To determine the size of a non-unique or a unique array, use `size` and `usize` respectively. Both return the size of the array in constant time.

Unique array updates `{`*array* `&` `[`*index*`]` `=` *expr*`}` are translated to `update` *array index expr*. A useful function is `createArray` which takes an integer that represents the number of elements an array should have, and a value for each of the elements. For instance, `createArray 1000 "Hello␣World!"` creates a unique array (so that it can be updated destructively later on) that consists of 1000 `"Hello␣World!"` text elements. The function `replace` is useful when working with arrays of unique elements: it updates the array at the given index position with the new value, and returns the old value at the same position.

7 Working with texts

As was already mentioned in Section 4, the `String` type is a basic type in Clean, even though it is a composite type, viz. an unboxed array of characters. There is special notation for convenient creation (`myString = "Hello␣World!"`), and there is a module, `StdString`, that contains useful operations on strings. Because strings are also arrays, everything you can do with arrays, you can also do with strings (see Section 6).

`StdString` provides string comparison (`==` and `<`) and the usual conversion instances. Strings can be sliced (`%`) and concatenated (`+++`). If you need to work with unique strings, then `+++.` can be used to produce a unique string for further destructive updates. You can update the ith element (counted from zero) of a string s with a new character c with the `:=` operator: s`:=(`i`,`c`)`. For instance,

```
Start = let hello = "Hello␣World!" in hello := (size hello-1,'?')
```

results in

```
Hello World?
```

Strings are arrays, so the following program produces the same result:

```
Start = {"Hello␣World!" & [11]='?'}
```

If you need to manipulate strings, it is in many case more convenient to use lists, simply because `StdList` offers a lot of list manipulating functions. `StdList` contains two functions for this conversion: the `fromString` and `toString` instances for character lists.

Module `StdCharList` offers a number of functions that aim at 'word processing' functionality. Usually you get a big chunk of text as a single string, containing newline characters which indicate line ends. Given such a string, you can easily convert it to a list of lines (where each line is a list of characters) with `mklines`. Note that `mklines` removes the trailing newline character from each line. You can then process the line elements, and glue the lines back together into a big string with `flatlines`, which inserts the appropriate newline characters back at the end of each line.

If you need to align lines of text to the left, right or center, given a certain column width, then you can use one of the functions `ljustify`, `rjustify`, or `cjustify` respectively. As a typical example, the following function transforms a string into another string in which all lines are centered for a certain column width:

```
centerAllLines :: Int -> String -> String
centerAllLines width
   = toString o flatlines o (map (cjustify width)) o mklines o fromString
```

Finally, the little utility function `spaces n` produces a list of `n` spaces (it is literally implemented as `spaces n = repeatn n '␣'`.

8 Working with tuples

`StdTuple` contains a number of functions for easy access to tuples and triplets. To select the first or second part of a tuple, use `fst` or `snd` respectively. To select the first or second or third

part of a triplet, use fst3 or snd3 or thd3 respectively. Tuples and triplets can be compared for equality (== instance) and lexical ordering (< instance). If you have two functions f and g, then they can be applied to the first and second component of a tuple with app2 (f,g). Use app3 (f,g,h) in case of triplets. Finally, there are two conversion functions to switch between a function of type (a,b) -> c to a b -> c (curry) and vice versa (uncurry).

9 StdFunctions

This module contains a number of standard higher order functions. Note that in Clean 2.x this module had a different name (StdFunc) and a few extra functions that have been moved to other modules in Clean 3.x. The remaining functions have not been changed.

The first two are id and const with trivial implementations: id x = x and const a b = a.

When using higher order functions, you sometimes notice that the function that you want to pass to such a function has the arguments in the wrong order. Suppose you have a function f of type a b -> c but need a function in which the argument order is flipped, hence of type b a -> c. Then flip f is a function of type b a -> c. Again, flip is implemented trivially as flip f b a = f a b.

The function composition operator, o, is the same as in high school: if you want to apply function g *after* function f, then you can this with g o f. Function composition is concisely defined as: (o) g f = \x = g (f x) (note that it must have arity two because it is an operator).

A function that you encounter in literature is twice, which is defined as twice f x = f (f x).

StdFunc contains a number of functions that resemble imperative control structures:

```
while :: !(a -> .Bool) (a -> a)   a ->  a
until :: !(a -> .Bool) (a -> a)   a ->  a
iter  :: !Int          (.a -> .a) .a -> .a
```

Given a predicate p, and a computation f on some arbitrary data type, and an initial value v_0, while p f v_0 applies f to subsequent values $v_i = f^i\ v_0$ while p holds for these values. The value returned is the first v_n for which p v_n is false. The computation until p f v_0 is basically the same, except that the computations are performed as long as p does not hold, and the value returned is the first v_n for which p v_n is true. The computation iter n f $v_0 = f^n\ v_0$, provided that n ≥ 0.

10 StdMaybe

The module StdMaybe defines a parameterized algebraic data type (Maybe) that is well known from literature when dealing with computations that might not always be able to return a meaningful answer of some type.

```
:: Maybe x = Just x | Nothing
```

This occurs more often than you might expect. For instance, here are a few candidate functions from StdEnv that could have used this type, but currently do not do this, mostly because of historic reasons. To avoid confusion, I add ‘ behind their function names:

```
hd‘                :: [a] -> Maybe a      // instead of hd (5.4.4)
tl‘                :: [a] -> Maybe [a]    // instead of tl (5.4.5)

class FileSystem‘ env where
      fopen‘       :: !{#Char} !Int !*env -> (!Maybe *File,!*env) // instead of fopen  (4.5.1)
      stdio‘       ::               !*env -> (!*File,      !*env) // instead of sfopen (4.5.1)
      ...

class fromInt‘    a :: !Int     -> Maybe a  // instead of fromInt    (2.1)
class fromChar‘   a :: !Char    -> Maybe a  // instead of fromChar   (2.1)
class fromBool‘   a :: !Bool    -> Maybe a  // instead of fromBool   (2.1)
class fromReal‘   a :: !Real    -> Maybe a  // instead of fromReal   (2.1)
class fromString‘ a :: !{#Char} -> Maybe a  // instead of fromString (2.1)
```

This example illustrates the role of using the `Maybe` type as a more explicit way of documenting the behaviour of functions: it is not possible to take the head element or the tail list of an empty list, a file can not always be opened, and conversions from one type to another also does not always have a result. For this reason, I recommend to use this type in your programs if you come across such situations.

The above example also illustrates that the approach works for both ‘ordinary’ values as well as uniquely attributed values (section 3.2).

To test whether a function has a meaningful value, the `isJust` predicate function can be used for ‘ordinary’ values, and `isJustU` for uniquely attributed values:

```
isJust  ::   !(Maybe .x) -> Bool
isJustU :: !u:(Maybe .x) -> (!Bool, !u:Maybe .x)
```

The `Bool` result of both functions is `True` in case the argument starts with the data constructor `Just`. To actually obtain the meaningful value, `fromJust` is used:

```
fromJust :: !(Maybe .x) -> .x
```

The functions `isNothing` and `isNothingU` test for the data constructor `Nothing`.

```
isNothing  ::   !(Maybe .x) -> Bool
isNothingU :: !u:(Maybe .x) -> (!Bool, !u:Maybe .x)
```

If equality is defined for the meaningful values (`==`, section 2.1) of type `a`, then you can compare values of type `Maybe a` as well:

```
instance == (Maybe x) | == x
```

The remaining functions work both for ‘ordinary’ values as well as uniquely attributed values. Similar to `map` in `StdList` (5.4.7), you can alter the meaningful value inside the `Just` data constructor with a function in one go:

```
mapMaybe :: .(.x -> .y) !(Maybe .x) -> Maybe .y
```

Finally, there are a few conversions functions to and from lists of values of type `a`:

```
maybeToList :: !(Maybe .a) -> [.a]
listToMaybe ::       ![.a] -> .Maybe .a
catMaybes   :: ![Maybe .a] -> .[.a]
```

The function maybeToList converts Nothing into [] and Just x into [x]. The function listToMaybe converts [] to Nothing, and [x : _] into Just x. Finally, catMaybes filters out all Just x values and removes the Just data constructor.

11 StdMisc

The smallest module of StdEnv is StdMisc, which defines two functions that are useful to indicate error-conditions in a program:

```
abort :: !{#Char} -> .a
undef ::             .a
```

As you can see from their types, they can be applied in any context. abort msg, when evaluated, prints msg and aborts the computation of the entire program, while undef simply aborts without an additional message. Use these functions to cover unexpected cases in your code.

12 StdDebug

The module StdDebug defines four functions that are very useful when trying to figure out what is going on in your program. An example of using trace_n has been shown in section 5.6.

```
trace    :: !msg .a -> .a | toString msg
trace_n  :: !msg .a -> .a | toString msg
trace_t  :: !msg -> Bool  | toString msg
trace_tn :: !msg -> Bool  | toString msg
```

All of these functions expect as first argument the message that has to be printed. The strictness annotation (!) indicates that this message is going to be evaluated. Any type will do, as long as an instance of the overloaded function toString is available in your program, and imported when using these functions. The function trace my_message expression first prints my_message on the console, and only then evaluates expression. Function trace_n does the same, but adds an extra newline after my_message, so you don't have to program that yourself. The function trace_t my_message first prints my_message on the console, and then evaluates to True. Function trace_tn does the same, but adds an extra newline after my_message, so you don't have to program that yourself.